αKanren

A Fresh Name in Nominal Logic Programming

William E. Byrd and Daniel P. Friedman

Department of Computer Science, Indiana University, Bloomington, IN 47408

{webyrd,dfried}@cs.indiana.edu

Abstract

We present αKanren, an embedding of nominal logic programming in Scheme. αKanren is inspired by αProlog and MLSOS, and allows programmers to easily write interpreters, type inferencers, and other programs that must reason about scope and binding. αKanren subsumes the functionality, syntax, and implementation of miniKanren, itself an embedding of logic programming in Scheme.

We present the *complete* implementation of αKanren, written in portable R^5RS Scheme. In addition to the implementation, we provide introductions to miniKanren and αKanren, and several example programs, including a type inferencer for the simply typed λ-calculus.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages

Keywords αKanren, αProlog, MLSOS, Scheme, miniKanren, logic programming, nominal logic, nominal unification

1. Introduction

We present a complete implementation in R^5RS Scheme of αKanren, which extends the miniKanren logic programming language (Byrd and Friedman 2006) with operators for *nominal logic programming*. αKanren was inspired by αProlog (Cheney 2004; Cheney and Urban 2004) and MLSOS (Lakin and Pitts 2007), and their use of nominal logic (Pitts 2003) to solve a class of problems more elegantly than is possible with conventional logic programming. The purpose of this paper is to present a readily understandable and hackable embedding of nominal logic programming in portable Scheme, along with a self-contained introduction to nominal logic programming.

Like αProlog and MLSOS, αKanren allows programmers to explicitly manage names and bindings, making it easier to write interpreters, type inferencers, and other programs that must reason about scope. αKanren also eases the burden of implementing a language from its structural operational semantics, since the requisite side-conditions can often be trivially encoded in nominal logic.

A standard class of such side conditions is to state that a certain variable name cannot occur free in a particular expression. It is a simple matter to check for free occurrences of a variable name in a fully-instantiated term, but in a logic program the term might contain unbound logic variables. At a later point in the program those variables might be instantiated to terms containing the variable name in question. Also, when the writer of semantics employs the equality symbol, the writer generally has in mind that two terms are the same, but what they really mean is that the two terms are the same *up to α-equivalence*, as in the variable hygiene convention popularized by Barendregt (1984). As functional programmers, we would never quibble with the statement: $\lambda x.x = \lambda y.y$, yet without the implicit assumption that one can rename variables using α-conversion, we would have to forego this obvious equality. And again, if either expression contains an unbound logic variable, it is impossible to perform a full parallel tree walk to determine if the two expressions are α-equivalent: at least part of the tree walk must be deferred until one or both expressions are fully instantiated.

We proceed as follows. Section 2 provides a brief review of miniKanren, an embedding of logic programming in Scheme; miniKanren is similar to standard Prolog in that it does not include nominal logic operators. In Section 3, we extend miniKanren to arrive at the αKanren language, and explain how these extensions correspond to the ideas of nominal logic programming. We give several example programs and their results, including non-naive β-substitution and a type inferencer for the simply typed λ-calculus. In Section 4 we define the nominal unifier, which is the heart of nominal logic programming. In the appendices we define and demonstrate a simple pattern matcher, followed by the remainder of the αKanren implementation.

Revised November, 2007, with earlier version appearing in Proceedings of the 2007 Workshop on Scheme and Functional Programming, available as Université Laval Technical Report DIUL-RT-0701

2. miniKanren Refresher

αKanren extends miniKanren, an embedding of logic programming in Scheme. In this section we briefly review the miniKanren language; readers already familiar with miniKanren can safely skip to Section 3, while those wishing to learn more about the language should see Byrd and Friedman (2006) (from which this refresher was adapted) and Friedman et al. (2005).

Our code uses the following typographic conventions. Lexical variables are in *italic*, forms are in **boldface**, and quoted symbols are in sans serif. By our convention, names of relations end with a superscript o—for example *subst*^{o}, which is entered as `substo`. Relational operators do not follow this convention: $\equiv$ (entered as `==`), **cond**^{e} (entered as `conde`), and **exist** (changed from **fresh**, since we use **fresh** in the next section). Similarly, (**run**^{5} (*q*) *body*) and (**run*** (*q*) *body*) are entered as `(run 5 (q) body)` and `(run* (q) body)`, respectively.

miniKanren extends Scheme with three operators: $\equiv$, **cond**^{e}, and **exist**. There is also **run**, which serves as an interface between Scheme and miniKanren, and whose value is a list.

exist, which syntactically looks like **lambda**, introduces new variables into its scope; $\equiv$ unifies two terms. Thus

```
(exist (x y z) (≡ x z) (≡ 3 y))
```

would associate x with z and y with 3. This, however, is not a legal miniKanren program—we must wrap a **run** around the entire expression.

```
(run^1 (q) (exist (x y z) (≡ x z) (≡ 3 y))) ⇒ (_0)
```

The value returned is a list containing the single value $\_{0}$; we say that $\_{0}$ is the *reified value* of the unbound variable q. q also remains unbound in

```
(run^1 (q) (exist (x y) (≡ x q) (≡ 3 y))) ⇒ (_0)
```

We can get back other values, of course.

```
(run^1 (y)        (run^1 (q)        (run^1 (y)
  (exist (x z)      (exist (x z)      (exist (x y)
    (≡ x z)           (≡ x z)           (≡ 4 x)
    (≡ 3 y)))         (≡ 3 z)           (≡ x y))
                      (≡ q x)))       (≡ 3 y))
```

Each of these examples returns (3); in the rightmost example, the y introduced by **exist** is different from the y introduced by **run**. **run** can also return the empty list, indicating that there are no values.

```
(run^1 (x) (≡ 4 3)) ⇒ ()
```

We use **cond**^{e} to get several values—syntactically, **cond**^{e} looks like **cond** but without ⇒ or **else**. For example,

```
(run^2 (q)
  (exist (x y z)
    (cond^e
      ((≡ `(,x ,y ,z ,x) q))
      ((≡ `(,z ,y ,x ,z) q))))) ⇒

((_0 _1 _2 _0) (_0 _1 _2 _0))
```

Although the two **cond**^{e} clauses are different, the values returned are identical. This is because distinct reified unbound logic variables are assigned distinct subscripts, increasing from left to right—the numbering starts over again from zero within each value, which is why the reified value of x is $\_{0}$ in the first value but $\_{2}$ in the second value.

The superscript 2 denotes the maximum length of the resultant list. If the superscript $*$ is used, then there is no maximum imposed. This can easily lead to infinite loops:

```
(run^* (q)
  (let loop ()
    (cond^e
      ((≡ #f q))
      ((≡ #t q))
      ((loop)))))
```

Had the $*$ been replaced by a non-negative integer n, then a list of n alternating `#f`'s and `#t`'s would be returned. The **cond**^{e} succeeds while associating q with `#f`, which accounts for the first value. When getting the second value, the second **cond**^{e}-clause is tried, and the association made between q and `#f` is forgotten—we say that q has been *renewed*. In the third **cond**^{e}-clause, q is renewed once again.

We now look at several interesting examples that rely on *any*^{o}, which tries g an unbounded number of times.

```
(define any^o
  (λ (g)
    (cond^e
      (g)
      ((any^o g)))))
```

Consider the first example

```
(run^* (q)
  (cond^e
    ((any^o (≡ #f q)))
    ((≡ #t q))))
```

which does not terminate because the call to *any*^{o} succeeds an unbounded number of times. If $*$ were replaced by 5, then we would get (`#t #f #f #f #f`). (The user should not be concerned with the order in which values are returned.)

Now consider

```
(run^10 (q)
  (any^o
    (cond^e
      ((≡ 1 q))
      ((≡ 2 q))
      ((≡ 3 q))))) ⇒

(1 2 3 1 2 3 1 2 3 1)
```

Here the values 1, 2, and 3 are interleaved; our use of *any*^{o} ensures that this sequence will be repeated indefinitely.

Even if some **cond**^{e}-clauses loop indefinitely, other **cond**^{e}-clauses can contribute to the values returned by a **run** expression. (We are not concerned with expressions looping indefinitely, however.) For example,

```
(run^3 (q)
  (let ((never^o (any^o (≡ #f #t))))
    (cond^e
      ((≡ 1 q))
      (never^o)
      ((cond^e
         ((≡ 2 q))
         (never^o)
         ((≡ 3 q)))))))
```

returns (1 2 3); replacing **run**^{3} with **run**^{4} would cause divergence, however, since there are only three values, and since *never*^{o} would loop indefinitely.

3. Introduction to αKanren

αKanren extends the miniKanren language with two additional operators, **fresh** and # (entered as `hash`), and one term constructor, ⋈ (entered as `tie`).

fresh, which syntactically looks like **exist**, introduces new *noms* into its scope. (Noms are also called “names” or “atoms”, overloaded terminology which we avoid.) Conceptually, a nom represents a variable name[1]; however, a nom behaves more like a constant than a variable, since it only unifies with itself or with an unbound variable.

```
(run* (q) (fresh (a) (≡ a a))) ⇒ (_0)
(run* (q) (fresh (a) (≡ a 5))) ⇒ ()
(run* (q) (fresh (a b) (≡ a b))) ⇒ ()
(run* (q) (fresh (b) (≡ b q))) ⇒ (b0)
```

A reified nom is subscripted in the same fashion as a reified variable, but the name with which the nom is declared is used instead of an underscore (_)—hence the (b_0) in the final example above. **fresh** forms can be nested, which may result in noms being shadowed.

```
(run* (q)
  (exist (x y z)
    (fresh (a)
      (≡ x a)
      (fresh (a b)
        (≡ y a)
        (≡ ‘(,x ,y ,z ,a ,b) q))))) ⇒
((a0 a1 _0 a1 b0))
```

Here a_0 and a_1 represent different noms, which will not unify with each other.

⋈ is a *term constructor* used to limit the scope of a nom within a term (any value that can contain variables and noms).

```
(define-syntax ⋈
  (syntax-rules ()
    ((_ a t) ‘(tie ,a ,t))))
```

Terms constructed using ⋈ are called *binders*. In the term created by the expression (⋈ *a t*), all occurrences of the nom *a* within term *t* are considered bound. We refer to the term *t* as the *body* of (⋈ *a t*), and to the nom *a* as being in *binding position*. The ⋈ constructor does not create noms; rather, it delimits the scope of noms, already introduced using **fresh**.

For example, consider this **run*** expression.

```
(run* (q)
  (fresh (a b)
    (≡ (⋈ a ‘(foo ,a 3 ,b)) q))) ⇒
((tie a0 (foo a0 3 b0)))
```

The tagged list (tie a_0 (foo a_0 3 b_0)) is the reified value of the term constructed using ⋈. (The tag name tie is a pun—the bowtie ⋈ is the “tie that binds.”) The nom whose reified value is a_0 occurs bound within the term (tie a_0 (foo a_0 3 b_0)) while b_0 occurs free in that same term.

# introduces a *freshness constraint* (henceforth referred to as simply a *constraint*). The expression (# *a t*) asserts that the nom *a* does *not* occur free in term *t*—if *a* occurs free in *t*, then (# *a t*) fails. Furthermore, if *t* contains an unbound variable *x*, and some later unification involving *x* results in *a* occurring free in *t*, then that unification fails.

[1] Less commonly, a nom represents a non-variable entity. For example, a nom may represent a channel name in the π-calculus—see Cheney (2004) for details.

```
(run* (q) (fresh (a) (≡ ‘(3 ,a #t) q) (# a q))) ⇒ ()
(run* (q) (fresh (a) (# a q) (≡ ‘(3 ,a #t) q))) ⇒ ()
(run* (q) (fresh (a b) (# a (⋈ b a)))) ⇒ ()
(run* (q) (fresh (a) (# a (⋈ a a)))) ⇒ (_0)
(run* (q)
  (exist (x y z)
    (fresh (a)
      (# a x)
      (≡ ‘(,y ,z) x)
      (≡ ‘(,x ,a) q)))) ⇒
((((_0 _1) a0) : ((a0 . _0) (a0 . _1))))
```

In the fourth example, the constraint (# *a* (⋈ *a a*)) is not violated because *a* does not occur free in (⋈ *a a*). In the final example, the partial instantiation of *x* causes the constraint introduced by (# *a x*) to be “pushed down” onto the unbound variables *y* and *z*. The answer comprises two parts, separated by a colon and enclosed in an extra set of parentheses: the reified value of ((*y z*) *a*) and a list of reified constraints indicating that *a* cannot occur free in either *y* or *z*.

The notion of a constraint is prominent in the standard definition of α-equivalence (Stoy 1979):

$$\lambda a.M \equiv_\alpha \lambda b.[b/a]M \text{ where } b \text{ does not occur free in } M.$$

In αKanren this constraint is expressed as (# *b M*). We shall revisit the connection between constraints and α-equivalence shortly.

We now extend the standard notion of unification to that of *nominal unification* (Urban et al. 2004), which equates α-equivalent binders. Consider this **run*** expression.

```
(run* (q) (fresh (a b) (≡ (⋈ a a) (⋈ b b)))) ⇒ (_0)
```

Although *a* and *b* are distinct noms, (≡ (⋈ *a a*) (⋈ *b b*)) succeeds. According to the rules of nominal unification, the binders (⋈ *a a*) and (⋈ *b b*) represent the same term, and therefore unify.

The reader may suspect that, as in the definition of α-equivalence given above, nominal unification uses substitution to equate binders

$$(\bowtie\ a\ a) \equiv_\alpha (\bowtie\ b\ [b/a]a)$$

however, this is not the case.

Unfortunately, naive substitution does not preserve α-equivalence of terms, as shown in the following example given by Urban et al. (2004). Consider the α-equivalent terms (⋈ *a b*) and (⋈ *c b*); replacing all free occurrences of *b* with *a* in both terms yields (⋈ *a a*) and (⋈ *c a*), which are no longer α-equivalent.

Rather than using capture-avoiding substitution to address this problem, nominal logic uses the simple and elegant notion of a *nom swap*. Instead of performing a unidirectional substitution of *a* for *b*, the unifier exchanges all occurrences of *a* and *b* within a term, regardless of whether those noms appear free, bound, or in the binding position of a ⋈-constructed binder. Applying the swap (*a b*) to (⋈ *a b*) and (⋈ *c b*) yields the α-equivalent terms (⋈ *b a*) and (⋈ *c a*).

When unifying (⋈ *a a*) and (⋈ *b b*) in the **run*** expression above, the nominal unifier first creates the swap (*a b*) containing the noms in the binding positions of the two terms. The unifier then applies this swap to (⋈ *a a*), yielding (⋈ *b b*) (or equivalently, applies the swap to (⋈ *b b*), yielding $(\bowtie a\ a)$). Obviously $(\bowtie b\ b)$ unifies with itself, according to the standard rules of unification, and thus the nominal unification succeeds.

Of course, the terms being unified might contain unbound variables. In the simple example

```
(run* (q) (fresh (a b) (≡ (⋈ a q) (⋈ b b)))) ⇒ (a_0)
```

the swap $(a\ b)$ can be applied to $(\bowtie b\ b)$, yielding $(\bowtie a\ a)$. The terms $(\bowtie a\ a)$ and $(\bowtie a\ q)$ are then unified, associating q with a. However, in some cases a swap cannot be performed until a variable has become at least partially instantiated. For example, in the first call to $\equiv$ in

```
(run* (q)
  (fresh (a b)
    (exist (x y)
      (≡ (⋈ a (⋈ a x)) (⋈ a (⋈ b y)))
      (≡ `(,x ,y) q))))
```

the unifier cannot apply the swap $(a\ b)$ to either x or y, since they are both unbound. (The unifier does not generate a swap for the outer binders, since they have the same nom in their binding positions.)

Nominal unification solves this problem by introducing the notion of a *suspension*, which is a record of *delayed swaps* that may be applied later. We represent a suspension using the susp data structure, which comprises a list of suspended swaps and a variable.

$$(\mathsf{susp}\ ((a_n\ b_n)\ \ldots\ (a_1\ b_1))\ x)$$

The swaps are deferred until the variable x is instantiated (at least partially); at this point the swaps are applied to the instantiated portion of the term associated with x. Swaps are applied from right to left; that is, the result of applying the swaps to a term t can be determined by first exchanging all occurrences of noms a_1 and b_1 within t, then exchanging a_2 and b_2 within the resulting term, and continuing in this fashion until finally exchanging a_n with b_n.

Now that we have the notion of a suspension, we can define equality on binders (adapted from Urban et al. 2004):

> $(\bowtie a\ M)$ and $(\bowtie b\ N)$ are α-equivalent if and only if a and b are the same nom and M is α-equivalent to N, or if $(\mathsf{susp}\ ((a\ b))\ M)$ is α-equivalent to N and $(\#\ b\ M)$.

The side condition $(\#\ b\ M)$ is necessary, since if b occurred free in M, then b would be inadvertently captured (and replaced with a) by the suspension $(\mathsf{susp}\ ((a\ b))\ M)$.

Having defined equality on binders, we can examine the result of the previous **run*** expression.

```
(run* (q)
  (fresh (a b)
    (exist (x y)
      (≡ (⋈ a (⋈ a x)) (⋈ a (⋈ b y)))
      (≡ `(,x ,y) q)))) ⇒
((((susp ((a_0 b_0)) _0) _0) : ((a_0 . _0))))
```

The first call to $\equiv$ applies the swap $(a\ b)$ to the unbound variable y, and then associates the resulting suspension $(\mathsf{susp}\ ((a\ b))\ y)$ with x. Of course, the unifier could have applied the swap to x instead of y, resulting in a symmetric answer. The freshness constraint states that the nom a can never occur free within y, as required by the definition of binder equivalence.

Here is a translation of a quiz presented in Urban et al. (2004), demonstrating some of the finer points of nominal unification.

```
(run* (q)
  (fresh (a b)
    (exist (x y)
      (cond^e
        ((≡ (⋈ a (⋈ b `(,x ,b))) (⋈ b (⋈ a `(,a ,x)))))
        ((≡ (⋈ a (⋈ b `(,y ,b))) (⋈ b (⋈ a `(,a ,x)))))
        ((≡ (⋈ a (⋈ b `(,b ,y))) (⋈ b (⋈ a `(,a ,x)))))
        ((≡ (⋈ a (⋈ b `(,b ,y))) (⋈ a (⋈ a `(,a ,x))))))
      (≡ `(,x ,y) q)))) ⇒
((a_0 b_0)
 (_0 (susp ((a_0 b_0)) _0))
 ((_0 (susp ((b_0 a_0)) _0)) : ((b_0 . _0))))
```

The first **cond**^{e} clause fails, since x cannot be associated with both a and b. The second clause succeeds, associating x with a and y with b. The third clause applies the swap $(a\ b)$ to $(\bowtie a\ (a\ x))$, yielding $(\mathsf{tie}\ b\ (b\ (\mathsf{susp}\ ((a\ b))\ x)))$. This term is then unified with $(\bowtie b\ (b\ y))$, associating y with the suspension $(\mathsf{susp}\ ((a\ b))\ x)$. The fourth clause should look familiar—it is similar to the previous **run*** expression.

We can interpret the successful unification of binders $(\bowtie a\ a)$ and $(\bowtie b\ b)$ as showing that the λ-calculus terms $\lambda a.a$ and $\lambda b.b$ are identical, up to α-equivalence. We need not restrict our interpretation to λ terms, however, since other scoping mechanisms have similar properties. For example, the same successful unification also shows that $\forall a.a$ and $\forall b.b$ are equivalent in first-order logic, and similarly, that $\exists a.a$ and $\exists b.b$ are equivalent.

We can tag terms in order to disambiguate their interpretation. For example, this program shows that $\lambda a.\lambda b.a$ and $\lambda c.\lambda d.c$ are equivalent.

```
(run* (q)
  (exist (t u)
    (fresh (a b c d)
      (≡ `(lam ,(⋈ a `(lam ,(⋈ b `(var ,a))))) t)
      (≡ `(lam ,(⋈ c `(lam ,(⋈ d `(var ,c))))) u)
      (≡ t u)))) ⇒
(_0)
```

Of course, not all λ-calculus terms are equivalent.

```
(run* (q)
  (exist (t u)
    (fresh (a b c d)
      (≡ `(lam ,(⋈ a `(lam ,(⋈ b `(var ,a))))) t)
      (≡ `(lam ,(⋈ c `(lam ,(⋈ d `(var ,d))))) u)
      (≡ t u)))) ⇒
()
```

Here $(\equiv t\ u)$ fails, showing that $\lambda a.\lambda b.a$ and $\lambda c.\lambda d.d$ are not α-equivalent.

3.1 Non-naive Substitution

We now consider a simple, but useful, nominal logic program adapted from Cheney and Urban (2004) that performs capture-avoiding substitution (that is, β-substitution). $subst^o$ implements the relation $[new/a]e = out$ where e, new, and out are tagged lists representing λ-calculus terms, and where a is a nom representing a variable name. (We refer the interested reader to Cheney and Urban for a full description of $subst^o$.)

```
(define subst^o
  (λ (e new a out)
    (cond^e
      ((≡ `(var ,a) e) (≡ new out))
      ((exist (y)
         (≡ `(var ,y) e)
         (≡ `(var ,y) out)
         (# a y)))
      ((exist (rator ratorres rand randres)
         (≡ `(app ,rator ,rand) e)
         (≡ `(app ,ratorres ,randres) out)
         (subst^o rator new a ratorres)
         (subst^o rand new a randres)))
      ((exist (body bodyres)
         (fresh (c)
           (≡ `(lam ,(⋈ c body)) e)
           (≡ `(lam ,(⋈ c bodyres)) out)
           (# c a)
           (# c new)
           (subst^o body new a bodyres)))))))
```

The first *subst^o* example shows that $[b/a]\lambda a.ab \equiv_\alpha \lambda c.cb$.

```
(run* (q)
  (fresh (a b)
    (subst^o
      `(lam ,(⋈ a `(app (var ,a) (var ,b)))) `(var ,b) a q))) ⇒
((lam (tie c_0 (app (var c_0) (var b_0)))))
```

Naive substitution would have produced $\lambda b.bb$ instead.

This second example shows that $[a/b]\lambda a.b \equiv_\alpha \lambda c.a$.

```
(run* (x)
  (fresh (a b)
    (subst^o `(lam ,(⋈ a `(var ,b))) `(var ,a) b x))) ⇒
((lam (tie c_0 (var a_0))))
```

Naive substitution would have produced $\lambda a.a$.

3.2 Type Inferencer for Simply Typed λ-calculus

Let us consider a second non-trivial αKanren example: a type inferencer for the simply typed λ-calculus (also adapted from Cheney and Urban 2004). *typ^o* relates a λ-calculus term *e* to its type *te* in type environment *g*.

```
(define typ^o
  (λ (g e te)
    (cond^e
      ((exist (x)
         (≡ `(var ,x) e)
         (lookup^o x te g)))
      ((exist (rator trator rand trand)
         (≡ `(app ,rator ,rand) e)
         (≡ `(→ ,trand ,te) trator)
         (typ^o g rator trator)
         (typ^o g rand trand)))
      ((exist (ê tê trand ĝ)
         (fresh (b)
           (≡ `(lam ,(⋈ b ê)) e)
           (≡ `(→ ,trand ,tê) te)
           (# b g)
           (≡ `((,b . ,trand) . ,g) ĝ)
           (typ^o ĝ ê tê)))))))
```

The *lookup^o* helper relation finds the type *tx* associated with the type variable *x* in the current type environment *g*.

```
(define lookup^o
  (λ (x tx g)
    (exist (a d)
      (≡ `(,a . ,d) g)
      (cond^e
        ((≡ `(,x . ,tx) a))
        ((exist (x̂ tx̂)
           (≡ `(,x̂ . ,tx̂) a)
           (# x x̂)
           (lookup^o x tx d)))))))
```

The first *typ^o* example shows that $\lambda c.\lambda d.c$ has type $(\alpha \to (\beta \to \alpha))$.

```
(run* (q)
  (fresh (c d)
    (typ^o '() `(lam ,(⋈ c `(lam ,(⋈ d `(var ,c))))) q))) ⇒
((→ _0 (→ _1 _0)))
```

The second example shows that self-application doesn't type check, since the nominal unifier uses the occurs check (Lloyd 1987).

```
(run* (q)
  (fresh (c)
    (typ^o '() `(lam ,(⋈ c `(app (var ,c) (var ,c)))) q))) ⇒
()
```

The final example is the most interesting, since it searches for terms that inhabit the type (int → int).

```
(run2 (q) (typ^o '() q '(→ int int))) ⇒
((lam (tie b_0 (var b_0)))
 (lam (tie b_0 (app (lam (tie b_1 (var b_1))) (var b_0)))))
```

The first two terms found are $\lambda b.b$ and $\lambda b.(\lambda a.a)b$.

This ends the introduction to αKanren. For additional simple examples of nominal logic programming, we suggest Cheney (2004), Cheney and Urban (2004), Urban et al. (2004), and Lakin and Pitts (2007), which are also excellent choices for understanding the theory of nominal logic. Next we discuss our implementation of nominal unification.

4. Nominal Unification

Nominal unification occurs in two distinct phases: the first processes equations, while the second processes constraints. The first phase takes a set of equations ϵ and transforms it into a substitution σ and a set of unresolved constraints δ. The second phase combines the unresolved constraints with the previously resolved constraints, which have both been brought up to date using *apply-subst*. Then, the unifier transforms these combined constraints into a set of resolved constraints ∇, and returns the list $(\sigma\ \nabla)$ as a *package*.

Nominal unification uses several data structures. A set of equations ϵ is represented as a list of pairs of terms. A substitution σ is represented as an association list of variables to terms. A set of constraints δ is represented as a list of pairs associating noms to terms; a ∇ is a δ in which all terms are unbound variables. In a substitution, a variable may have at most one association. In a δ (and therefore in a ∇) a nom may have multiple associations.

We represent a variable as a suspension containing an empty list of swaps. Several functions reconstruct suspensions that represent variables. However, our implementation of nominal unification assumes that variables can be compared using *eq?*.

In order to ensure that a variable is always *eq?* to itself, regardless of how many times it is reconstructed, we use a **letrec** trick: a suspension representing a variable contains a procedure of zero arguments (a *thunk*) that, when invoked, returns the suspension, thus maintaining the desired *eq?*-ness property. (In the text we conflate variables with their associated thunks.)

```
(define var
  (λ ()
    (letrec ((s (list 'susp '() (λ () s))))
      s)))
```

unify attempts to solve a set of equations ϵ in the context of a package (σ ∇). *unify* applies σ to ϵ, and then calls *apply-σ-rules* on the resulting set of equations. *apply-σ-rules* either successfully completes the first phase of nominal unification by returning a new σ and δ, or invokes the failure continuation *fk*, a jump-out continuation similar to Lisp's *catch* (Steele Jr. 1990).

```
(define unify
  (λ (ϵ σ ∇ fk)
    (let ((ϵ (apply-subst σ ϵ)))
      (mv-let ((σ̂ δ) (apply-σ-rules ϵ fk))
        (unify# δ (compose-subst σ σ̂) ∇ fk)))))
```

mv-let, defined in Appendix B, deconstructs a list of values.

In the second phase of nominal unification, *unify#* calls *apply-subst* to bring ∇ and δ up to date, then passes their union to *apply-∇-rules*.

```
(define unify#
  (λ (δ σ ∇ fk)
    (let ((δ (apply-subst σ δ))
          (∇ (apply-subst σ ∇)))
      (let ((δ (δ-union ∇ δ)))
        `(,σ ,(apply-∇-rules δ fk))))))
```

apply-σ-rules is a recursive function whose only task is to combine results returned by *σ-rules*. *σ-rules* takes two arguments: a single equation and the rest of the equations. If *σ-rules* fails, then *apply-σ-rules* invokes *fk*, and the result of *unify* is #f. Each successful call to *σ-rules* returns a new set of equations ϵ, a new σ, and a set of (unresolved) constraints δ. Successive calls to *σ-rules* resolve the equations in ϵ until there are no equations left.

```
(define apply-σ-rules
  (λ (ϵ fk)
    (cond
      ((null? ϵ) `(,empty-σ ,empty-δ))
      (else
       (let ((eqn (car ϵ)) (ϵ (cdr ϵ)))
         (mv-let ((ϵ σ δ) (or (σ-rules eqn ϵ) (fk)))
           (mv-let ((σ̂ δ̂) (apply-σ-rules ϵ fk))
             `(,(compose-subst σ σ̂) ,(δ-union δ̂ δ)))))))))
```

apply-∇-rules is similar to *apply-σ-rules*, but takes constraints instead of equations, and combines the results returned by *∇-rules*.

```
(define apply-∇-rules
  (λ (δ fk)
    (cond
      ((null? δ) empty-∇)
      (else
       (let ((c (car δ)) (δ (cdr δ)))
         (mv-let ((δ ∇) (or (∇-rules c δ) (fk)))
           (δ-union ∇ (apply-∇-rules δ fk))))))))
```

empty-σ, *empty-δ*, and *empty-∇* are defined in Appendix A.

In both *σ-rules* and *∇-rules* we use *untagged?* to distinguish untagged pairs from specially tagged pairs that represent binders, noms, and suspensions.

```
(define untagged?
  (λ (x)
    (not (memv x '(tie nom susp)))))
```

Here are the transformation rules of the nominal unification algorithm, derived from the rules in Urban et al. (2004). (*σ-rules* relies on **pmatch**, which is defined in Appendix B.)

```
(define σ-rules
  (λ (eqn ϵ)
    (pmatch eqn
      ((,c . ,ĉ)
       (guard (not (pair? c)) (equal? c ĉ))
       `(,ϵ ,empty-σ ,empty-δ))
      (((tie ,a ,t) . (tie ,â ,t̂))
       (guard (eq? a â))
       `(((,t . ,t̂) . ,ϵ) ,empty-σ ,empty-δ))
      (((tie ,a ,t) . (tie ,â ,t̂))
       (guard (not (eq? a â)))
       (let ((û (apply-π `((,a ,â)) t̂)))
         `(((,t . ,û) . ,ϵ) ,empty-σ ((,a . ,t̂)))))
      (((nom _) . (nom _))
       (guard (eq? (car eqn) (cdr eqn)))
       `(,ϵ ,empty-σ ,empty-δ))
      (((susp ,π ,x) . (susp ,π̂ ,x̂))
       (guard (eq? (x) (x̂)))
       (let ((δ (map (λ (a) `(,a . ,(x)))
                     (disagreement-set π π̂))))
         `(,ϵ ,empty-σ ,δ)))
      (((susp ,π ,x) . ,t)
       (guard (not (occurs√ (x) t)))
       (let ((σ `((,(x) . ,(apply-π (reverse π) t)))))
         `(,(apply-subst σ ϵ) ,σ ,empty-δ)))
      ((,t . (susp ,π ,x))
       (guard (not (occurs√ (x) t)))
       (let ((σ `((,(x) . ,(apply-π (reverse π) t)))))
         `(,(apply-subst σ ϵ) ,σ ,empty-δ)))
      (((,t_1 . ,t_2) . (,t̂_1 . ,t̂_2))
       (guard (untagged? t_1) (untagged? t̂_1))
       `(((,t_1 . ,t̂_1) (,t_2 . ,t̂_2) . ,ϵ) ,empty-σ ,empty-δ))
      (else #f))))
```

Clauses two and three in *σ-rules* implement α-equivalence of binders, as defined in Section 3. Clause five unifies two suspensions that have the same variable; in this case, *σ-rules* creates as many new freshness constraints as there are noms in the *disagreement set* (defined below) of the suspensions' swaps. Clauses six and seven are similar: each clause unifies a suspension containing a variable x and a list of swaps π with a term t. *σ-rules* creates a substitution associating x with the result of applying the swaps in π to t in *reverse order*, with the newest swap in π applied first. This substitution is applied to the context ϵ.

apply-π, below, applies a list of swaps π to a term v.

```
(define apply-π
  (λ (π v)
    (pmatch v
      (,c (guard (not (pair? c))) c)
      ((tie ,a ,t) (⋈ (apply-π π a) (apply-π π t)))
      ((nom _)
       (let loop ((v v) (π π))
         (if (null? π) v (apply-swap (car π) (loop v (cdr π))))))
      ((susp ,π̂ ,x)
       (let ((π `(,@π . ,π̂)))
         (if (null? π) (x) `(susp ,π ,x))))
      ((,a . ,d) `(,(apply-π π a) . ,(apply-π π d))))))
```

If v is a nom, then π's swaps are applied, with the oldest swap applied first. If v is a suspension with a list of swaps $\hat{\pi}$ and variable x, then the swaps in π are added to the swaps in $\hat{\pi}$. If this list is empty, then x's suspension is returned; otherwise, a new suspension is created with those swaps.

```
(define apply-swap
  (λ (swap a)
    (pmatch swap
      ((,a1 ,a2)
       (cond
         ((eq? a a2) a1)
         ((eq? a a1) a2)
         (else a))))))
```

The *∇-rules* are much simpler than the *σ-rules*. In the second clause, the nom $\hat{a}$ in the binding position of the binder is the same as a, so a can never appear free in t. In the fifth clause, the list of swaps π in the suspension are applied, in reverse order, to the nom a, yielding another nom. *∇-rules* then adds a new constraint associating this nom with the suspension's variable.

```
(define ∇-rules
  (λ (d δ)
    (pmatch d
      ((,a . ,c)
       (guard (not (pair? c)))
       `(,δ ,empty-∇))
      ((,a . (tie ,â ,t))
       (guard (eq? â a))
       `(,δ ,empty-∇))
      ((,a . (tie ,â ,t))
       (guard (not (eq? â a)))
       `(((,a . ,t) . ,δ) ,empty-∇))
      ((,a . (nom _))
       (guard (not (eq? a (cdr d))))
       `(,δ ,empty-∇))
      ((,a . (susp ,π ,x))
       `(,δ ((,(apply-π (reverse π) a) . ,(x)))))
      ((,a . (,t1 . ,t2))
       (guard (untagged? t1))
       `(((,a . ,t1) (,a . ,t2) . ,δ) ,empty-∇))
      (else #f))))
```

Finding the disagreement set of two lists of swaps π and $\hat{\pi}$ requires forming a set of all the noms in those lists, then applying both π and $\hat{\pi}$ to each nom a in this set. If (*apply-π* π a) and (*apply-π* $\hat{\pi}$ a) produce different noms, then a is in the *dis*agreement set. (*filter* and *remove-duplicates* are defined in Appendix A.4.)

```
(define disagreement-set
  (λ (π π̂)
    (filter
      (λ (a) (not (eq? (apply-π π a) (apply-π π̂ a))))
      (remove-duplicates
        (append (apply append π) (apply append π̂))))))
```

The *occurs*$^{\checkmark}$ is what one might expect.

```
(define occurs√
  (λ (x v)
    (pmatch v
      (,c (guard (not (pair? c))) #f)
      ((tie _ ,t) (occurs√ x t))
      ((nom _) #f)
      ((susp _ ,x̂) (eq? (x̂) x))
      ((,x̂ . ,ŷ) (or (occurs√ x x̂) (occurs√ x ŷ)))
      (else #f))))
```

4.1 Substitutions

compose-subst's definition is taken from Lloyd (1987). It takes two substitutions σ and τ, and constructs a new substitution $\hat{\sigma}$ in which each association (x . v) in σ is replaced by (x . $\hat{v}$), where $\hat{v}$ is the result of applying τ to v. Any association in τ whose variable has an association in $\hat{\sigma}$ is then filtered from τ. Also, any association of the form (x . x) is filtered from $\hat{\sigma}$. These filtered substitutions are then appended.

```
(define compose-subst
  (λ (σ τ)
    (let ((σ̂ (map
               (λ (a) `(,(car a) . ,(apply-subst τ (cdr a))))
               σ)))
      (append
        (filter (λ (a) (not (assq (car a) σ̂))) τ)
        (filter (λ (a) (not (eq? (car a) (cdr a)))) σ̂)))))
```

Next we define *apply-subst*. In the suspension case, *apply-subst* applies the list of swaps π to a variable, or to its binding.

```
(define apply-subst
  (λ (σ v)
    (pmatch v
      (,c (guard (not (pair? c))) c)
      ((tie ,a ,t) `(tie ,a ,(apply-subst σ t)))
      ((nom _) v)
      ((susp ,π ,x) (apply-π π (get (x) σ)))
      ((,x . ,y) `(,(apply-subst σ x) . ,(apply-subst σ y))))))
```

get, which is defined in Appendix A.4, finds the binding of a variable in a substitution or returns the variable if no binding exists.

4.2 *δ-union*

Finally we define *δ-union*, which forms the union of two δ's.

```
(define δ-union
  (λ (δ δ̂)
    (pmatch δ
      (() δ̂)
      ((,d . ,δ) (if (term-member? d δ̂)
                     (δ-union δ δ̂)
                     (cons d (δ-union δ δ̂)))))))

(define term-member?
  (λ (v v*)
    (pmatch v*
      (() #f)
      ((,v̂ . ,v*)
       (or (term-equal? v̂ v) (term-member? v v*))))))

(define term-equal?
  (λ (u v)
    (pmatch `(,u ,v)
      ((,c ,ĉ) (guard (not (pair? c)) (not (pair? ĉ)))
       (equal? c ĉ))
      (((tie ,a ,t) (tie ,â ,t̂))
       (and (eq? a â) (term-equal? t t̂)))
      (((nom _) (nom _)) (eq? u v))
      (((susp ,π ,x) (susp ,π̂ ,x̂))
       (and (eq? (x) (x̂)) (null? (disagreement-set π π̂))))
      (((,x . ,y) (,x̂ . ,ŷ))
       (and (term-equal? x x̂) (term-equal? y ŷ)))
      (else #f))))
```

Recall that δ denotes a set of unresolved constraints, where a constraint is a pair of a nom a and a term t. *δ-union* uses *term-member?*, which uses *term-equal?* when comparing two constraints. The definition of *term-equal?* is straightforward except when comparing two suspensions, in which case their variables must be the same, and the disagreement set of their lists of swaps must be empty.

5. Conclusion

αKanren and αProlog, and perhaps to a lesser extent, MLSOS, are all based on the nominal logic programming paradigm. Nominal programming languages based on functional programming include FreshML (Shinwell et al. 2003) and Cαml (Pottier 2006).

We have presented an implementation of αKanren, an embedding of nominal logic programming in Scheme. αKanren allows programmers to easily write interpreters, type inferencers, and other programs that must reason about scope and binding. αKanren subsumes the functionality, syntax, and implementation of miniKanren, itself an embedding of logic programming in Scheme. Our goal in creating αKanren has been to make available a simple, concise, and readily understandable implementation of a nominal logic programming language. We hope that others will be inspired to use and extend αKanren, and in the process become familiar with the powerful tools provided by nominal logic.

Acknowledgments

The original miniKanren implementation was developed with Oleg Kiselyov and Chung-chieh Shan, with additional tweaking by Mitch Wand; the code in Appendix A is based on that of miniKanren. We are grateful to Oleg Kiselyov for his **pmatch** pattern matcher. We thank Christian Urban and James Cheney for their insightful comments on an early draft of this paper, and Joe Near for reading the final draft. We owe our interest in nominal logic to Roshan James, Michael Adams, and Amr Sabry. We also thank the anonymous referees for their helpful remarks, and Danny Dubé for encouraging us to submit to the Scheme Workshop. Once again, we have found Dorai Sitaram's excellent SLaTeX package invaluable for typesetting our code.

References

Henk Barendregt. *The Lambda Calculus, its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.

William E. Byrd and Daniel P. Friedman. From variadic functions to variadic relations: A miniKanren perspective. In Robby Findler, editor, *Proceedings of the 2006 Scheme and Functional Programming Workshop*, University of Chicago Technical Report TR-2006-06, pages 105–117, 2006.

James Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, August 2004.

James Cheney and Christian Urban. αProlog: A logic programming language with names, binding and α-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming, ICLP 2004*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283, Saint-Malo, France, September 6–10, 2004. Springer.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming 2000*, September 5, 2000.

Ralf Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, Montreal, Canada, September 18–21, 2000*, pages 186–197. ACM Press, 2000.

Matthew R. Lakin and Andrew M. Pitts. A metalanguage for structural operational semantics. In Marco T. Morazán and Henrik Nilsson, editors, *Draft Proceedings of 8th Symposium on Trends in Functional Programming*, pages 1–16, 2007.

John Wylie Lloyd. *Foundations of logic programming*. Springer Verlag, New York, second extended edition, 1987.

David B. MacQueen, Philip Wadler, and Walid Taha. How to add laziness to a strict language without even being odd. In *Proceedings of the 1998 ACM Workshop on ML*, pages 24–30, September 1998. Baltimore, MD.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

François Pottier. *Cαml Reference Manual*. INRIA, 2006-12-14 edition, December 2006.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. FreshML: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25–29, 2003*, pages 263–274. ACM Press, 2003.

Guy L. Steele Jr. *COMMON LISP: The language*. Digital Press, second edition, 1990.

Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1979.

Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.

Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In Jean-Pierre Jouannaud, editor, *Proceedings of the Second Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128, Nancy, France, September 16–19, 1985. Springer-Verlag.

Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January, 1992. ACM Press.

A. αKanren Implementation

Our αKanren implementation comprises three kinds of operators: the interface operator **run**; goal constructors ≡, #, **cond**^{e}, **exist**, and **fresh**, which take a package *implicitly*; and functions such as *reify*, and the already defined *unify* and *unify#*, which take a package *explicitly*.

A goal g is a function that maps a package p to an ordered sequence p^∞ of zero or more packages. (For clarity, we notate λ as λ_G when creating such a function g.)

```
(define-syntax λG
  (syntax-rules () ((_ (p) e) (λ (p) e))))
```

Because a sequence of packages may be infinite, we represent it not as a list but as a p^∞, a special kind of stream that can contain either zero, one, or more packages (Hinze 2000; Wadler 1985). We use #f to represent the empty stream of packages. If p is a package, then p itself represents the stream containing just p. To represent a stream containing multiple packages, we use (**choice** p f), where p is the first package in the stream, and where f is a thunk that, when invoked, produces the remainder of the stream. (For clarity, we notate λ as λ_F when creating such a function f.) To represent an incomplete stream, we use (**inc** e), where e is an *expression* that evaluates to a p^∞—thus **inc** creates an f.

```
(define-syntax λF
  (syntax-rules () ((_ () e) (λ () e))))

(define-syntax choice
  (syntax-rules () ((_ a f) (cons a f))))

(define-syntax inc
  (syntax-rules () ((_ e) (λF () e))))
```

A singleton stream p is the same as (**choice** p (λ_F () #f)). However, for the goals that return only a single package, using this special representation of a singleton stream avoids the cost of unnecessarily building and taking apart pairs, and creating and invoking thunks.

To ensure that the values produced by these four kinds of p^∞'s can be distinguished, we assume that a package is never #f, a function, or a pair whose *cdr* is a function. To discriminate among these four cases, we define **case**$^\infty$.

```
(define-syntax case∞
  (syntax-rules ()
    ((_ a∞ (() e0) ((f̂) e1) ((â) e2) ((a f) e3))
     (pmatch a∞
       (#f e0)
       (,f̂ (guard (procedure? f̂)) e1)
       (,â (guard (not
                    (and (pair? â)
                         (procedure? (cdr â)))))
         e2)
       ((,a . ,f) e3)))))
```

The interface operator **run** uses *take* (defined below) to convert an f to an *even* stream (MacQueen et al. 1998). The definition of **run** places an artificial goal at the tail of g_0 g ...; this artificial goal reifies the variable x using the final package p produced by running the goals g_0 g ... in the empty package ‘(,*empty-σ* ,*empty-∇*).

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n (λF ()
               ((exist (x) g0 g ...
                  (λG (p) (cons (reify x p) '())))
                `(,empty-σ ,empty-∇)))))))
```

Wrapping the reified value in a list allows #f to appear as a value.

If the first argument to *take* is #f, then *take* returns the entire stream of reified values as a list, thereby providing the behavior of **run***. The **and** expressions within *take* detect this #f case.

```
(define take
  (λ (n f)
    (if (and n (zero? n))
      '()
      (case∞ (f)
        (() '())
        ((f) (take n f))
        ((a) a)
        ((a f) (cons (car a)
                 (take (and n (− n 1)) f)))))))
```

run* is trivially defined in terms of **run**.

```
(define-syntax run*
  (syntax-rules ()
    ((_ (x) g0 g ...)
     (run #f (x) g0 g ...))))
```

We represent the empty substitution, along with the empty unresolved and resolved constraint sets, as the empty list.

```
(define empty-σ '())

(define empty-δ '())

(define empty-∇ '())
```

A.1 Goal Constructors

The simplest goal constructors are ≡ and #; the goals they create return either a singleton stream or an empty stream.

```
(define ≡
  (λ (u v)
    (unifier unify `((,u . ,v)))))

(define #
  (λ (a t)
    (unifier unify# `((,a . ,t)))))

(define unifier
  (λ (fn set)
    (λG (p)
      (mv-let ((σ ∇) p)
        (call/cc (λ (fk) (fn set σ ∇ (λ () (fk #f)))))))))
```

To take the conjunction of goals, we define **exist**, a goal constructor that first lexically binds variables built by *var*, and then combines successive goals using **bind***. The goal constructor **fresh** is identical to **exist**, except that it lexically binds noms instead of variables.

```
(define-syntax exist
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (p)
       (inc
         (let ((x (var)) ...)
           (bind* (g0 p) g ...)))))))

(define-syntax fresh
  (syntax-rules ()
    ((_ (a ...) g0 g ...)
     (λG (p)
       (inc
         (let ((a (nom 'a)) ...)
           (bind* (g0 p) g ...)))))))
```

```
(define nom
  (λ (a)
    `(nom ,(symbol→string a))))
```

bind* is short-circuiting: since the empty stream is represented by #f, any failed goal causes **bind*** to immediately return #f. **bind*** relies on *bind* (Moggi 1991; Wadler 1992), which applies the goal g to each element in the stream p^∞. The resulting p^∞'s are then merged using *mplus*, which combines a p^∞ and an f to yield a single p^∞. (*bind* is similar to Lisp's *mapcan* but uses *mplus* (not *append*) to interleave the values of streams.)

```
(define-syntax bind*
  (syntax-rules ()
    ((_ e) e)
    ((_ e g0 g ...)
     (let ((a∞ e))
       (and a∞ (bind* (bind a∞ g0) g ...))))))

(define bind
  (λ (a∞ g)
    (case∞ a∞
      (() #f)
      ((f) (inc (bind (f) g)))
      ((a) (g a))
      ((a f) (mplus (g a) (λF () (bind (f) g)))))))

(define mplus
  (λ (a∞ f)
    (case∞ a∞
      (() (f))
      ((f̂) (inc (mplus (f) f̂)))
      ((a) (choice a f))
      ((a f̂) (choice a (λF () (mplus (f) f̂)))))))
```

To take the disjunction of goals we define **cond**^{e}, a goal constructor that combines successive **cond**^{e}-lines using **mplus***, which in turn relies on *mplus*. We use the same implicit package p for each **cond**^{e}-line. To avoid unwanted divergence, we treat the **cond**^{e}-lines as a single **inc** stream.

```
(define-syntax conde
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (p)
       (inc (mplus* (bind* (g0 p) g ...)
                    (bind* (g1 p) ĝ ...)
                    ...))))))

(define-syntax mplus*
  (syntax-rules ()
    ((_ e) e)
    ((_ e0 e ...) (mplus e0 (λF () (mplus* e ...))))))
```

A.2 Reification

reify takes a variable x and a package p, and returns the value associated with x in p (along with any relevant constraints), first replacing all variables and noms with symbols representing those entities. A constraint $(a\ .\ y)$ is *relevant* if both a and y appear in the value associated with x. We call this process of turning an αKanren value into a Scheme value *reification*.

The first **cond** clause in the definition of *refiy* below returns only the reified value associated with x, when there are no relevant constraints. The **else** clause returns both the reified value of x and the reified set of relevant constraints; we have arbitrarily chosen the colon ':' to separate the reified value from the list of reified constraints.

```
(define reify
  (λ (x p)
    (mv-let ((σ ∇) p)
      (let* ((v (get x σ)) (s (reify-s v)) (v (walk* v s)))
        (let ((∇ (filter (λ (a)
                           (and (symbol? (car a))
                                (symbol? (cdr a))))
                   (walk* ∇ s))))
          (cond
            ((null? ∇) v)
            (else `(,v : ,∇))))))))
```

reify-s is the heart of the reifier. *reify-s* takes an arbitrary value v, and returns a substitution that maps every distinct nom and variable in v to a unique symbol. The trick to maintaining left-to-right ordering of the subscripts on these symbols is to process v from left to right, as can be seen in the last **pmatch** clause. When *reify-s* encounters a nom or variable, it determines if we already have a mapping for that entity. If not, *reify-s* extends the substitution with an association between the nom or variable and a new, appropriately subscripted symbol.

```
(define reify-s
  (letrec
    ((r-s (λ (v s)
            (pmatch v
              (,c (guard (not (pair? c))) s)
              ((tie ,a ,t) (r-s t (r-s a s)))
              ((nom ,n)
               (cond
                 ((assq v s) s)
                 (else
                   (let ((ok-nom? (nom-names-eq? n)))
                     (let ((rn (reify-n n ok-nom? s)))
                       (cons `(,v . ,rn) s))))))
              ((susp () _)
               (cond
                 ((assq v s) s)
                 (else
                  (let ((rn (reify-n "_" var? s)))
                    (cons `(,v . ,rn) s)))))
              ((susp ,π _) (r-s π s))
              ((,a . ,d) (r-s d (r-s a s)))))))
    (λ (v)
      (r-s v '()))))
```

walk* applies a special substitution s, which maps noms and variables to symbols, to an arbitrary value v.

```
(define walk*
  (λ (v s)
    (pmatch v
      (,c (guard (not (pair? c))) c)
      ((tie ,a ,t) `(tie ,(get a s) ,(walk* t s)))
      ((nom _) (get v s))
      ((susp () _) (get v s))
      ((susp ,π ,x) `(susp ,(walk* π s) ,(get (x) s)))
      ((,a . ,d) `(,(walk* a s) . ,(walk* d s))))))

(define nom-names-eq?
  (λ (n)
    (λ (x)
      (pmatch x
        ((nom ,m) (string=? m n))
        (else #f)))))
```

```
(define var?
  (λ (x)
    (pmatch x
      ((susp () _) #t)
      (else #f))))
```

reify-n returns a symbol representing an individual variable or nom; this symbol always ends with a period followed by a non-negative integer.

```
(define reify-n
  (λ (str pred s)
    (string→symbol
      (string-append str "."
        (cond
          ((assp pred s) ⇒
           (λ (p)
             (number→string (+ (num-of (cdr p)) 1))))
          (else "0"))))))
```

reify-n uses *assp* and *num-of*; *assp* is like *assq*, except it is well defined for any unary predicate, while (*num-of* 'abc.52) returns the "subscript" 52; *num-of* relies on the assumption that symbols representing noms do not contain periods.

```
(define num-of
  (λ (a)
    (let ((c* (memv #\. (string→list (symbol→string a)))))
      (string→number (list→string (cdr c*))))))
```

A.3 Impure Control Operators:

For completeness, we define three additional αKanren goal constructors not used in this paper: **project**, which can be used to access the values of variables, and **cond**^{a} and **cond**^{u}, which can be used to prune the search tree of a program. The examples from chapter 10 of *The Reasoned Schemer* (Friedman et al. 2005) demonstrate how **cond**^{a} and **cond**^{u} can be useful, and the pitfalls that await the unsuspecting reader.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g_0 g ...)
     (λ_G (p)
       (mv-let ((σ ∇) p)
         (let ((x (get x σ)) ...)
           (bind* (g_0 p) g ...)))))))

(define-syntax cond^a
  (syntax-rules ()
    ((_ (g_0 g ...) (g_1 ĝ ...) ...)
     (λ_G (p)
       (inc (if^a ((g_0 p) g ...) ((g_1 p) ĝ ...) ...))))))

(define-syntax if^a
  (syntax-rules ()
    ((_) #f)
    ((_ (e g ...) b ...)
     (let loop ((a^∞ e))
       (case^∞ a^∞
         (() (if^a b ...))
         ((f) (inc (loop (f))))
         ((a) (bind* a^∞ g ...))
         ((a f) (bind* a^∞ g ...)))))))

(define-syntax cond^u
  (syntax-rules ()
    ((_ (g_0 g ...) (g_1 ĝ ...) ...)
     (λ_G (p)
       (inc (if^u ((g_0 p) g ...) ((g_1 p) ĝ ...) ...))))))

(define-syntax if^u
  (syntax-rules ()
    ((_) #f)
    ((_ (e g ...) b ...)
     (let loop ((a^∞ e))
       (case^∞ a^∞
         (() (if^u b ...))
         ((f) (inc (loop (f))))
         ((a) (bind* a^∞ g ...))
         ((a f) (bind* a g ...)))))))
```

A.4 Familiar Helpers

The auxiliaries below are generally useful functions that might be part of any library.

```
(define get
  (λ (x s)
    (cond
      ((assq x s) ⇒ cdr)
      (else x))))

(define assp
  (λ (p s)
    (cond
      ((null? s) #f)
      ((p (car (car s))) (car s))
      (else (assp p (cdr s))))))

(define filter
  (λ (p s)
    (cond
      ((null? s) '())
      ((p (car s)) (cons (car s) (filter p (cdr s))))
      (else (filter p (cdr s))))))

(define remove-duplicates
  (λ (s)
    (cond
      ((null? s) '())
      ((memq (car s) (cdr s)) (remove-duplicates (cdr s)))
      (else (cons (car s) (remove-duplicates (cdr s)))))))
```

B. pmatch

In this appendix we describe **pmatch**, a simple pattern matcher written by Oleg Kiselyov. Let us first consider a simple example of **pmatch**.

```
(define h
  (λ (x y)
    (pmatch `(,x . ,y)
      ((,a . ,b) (guard (number? a) (number? b)) (+ a b))
      ((_ . ,c) (guard (number? c)) (* c c))
      (else (* x x)))))

(list (h 1 2) (h 'w 5) (h 6 'w)) ⇒ (3 25 36)
```

In this example, a dotted pair is matched against three different kinds of patterns.

In the first pattern, the value of x is lexically bound to a and the value of y is lexically bound to b. Before the pattern match succeeds, however, an optional guard is run within the scope of a and b. The guard succeeds only if x and y are numbers; if so, then the sum of x and y is returned.

The second pattern matches against a pair, provided that the optional guard succeeds. If so, the value of y is lexically bound to c, and the square of y is returned.

If y is not a number, then the third and final clause is tried. An **else** pattern matches against *any* value, and never includes a guard. In this case, the clause returns the square of x, which must be a number in order to avoid an error at runtime.

Below is the definition of **pmatch**, which is implemented using continuation-passing-style macros (Hilsdale and Friedman 2000).

```
(define-syntax pmatch
  (syntax-rules (else guard)
    ((_ (op arg ...) cs ...)
     (let ((v (op arg ...)))
       (pmatch v cs ...)))
    ((_ v) (if #f #f))
    ((_ v (else e0 e ...)) (begin e0 e ...))
    ((_ v (pat (guard g ...) e0 e ...) cs ...)
     (let ((fk (λ () (pmatch v cs ...))))
       (ppat v pat
         (if (and g ...) (begin e0 e ...) (fk))
         (fk))))
    ((_ v (pat e0 e ...) cs ...)
     (let ((fk (λ () (pmatch v cs ...))))
       (ppat v pat (begin e0 e ...) (fk))))))
```

```
(define-syntax ppat
  (syntax-rules (_ quote unquote)
    ((_ v _ kt kf) kt)
    ((_ v () kt kf) (if (null? v) kt kf))
    ((_ v (quote lit) kt kf)
     (if (equal? v (quote lit)) kt kf))
    ((_ v (unquote var) kt kf) (let ((var v)) kt))
    ((_ v (x . y) kt kf)
     (if (pair? v)
       (let ((vx (car v)) (vy (cdr v)))
         (ppat vx x (ppat vy y kt kf) kf))
       kf))
    ((_ v lit kt kf) (if (equal? v (quote lit)) kt kf))))
```

The first clause ensures that the expression whose value is to be **pmatch**ed against is evaluated only once. The second clause returns an unspecified value if no other clause matches.

The remaining clauses represent the three types of patterns supported by **pmatch**. The first is the trivial **else** clause, which matches against any datum, and which behaves identically to an **else** clause in a **cond** expression. The other two clauses are identical, except that the first one includes a guard containing one or more expressions—if the datum matches against the pattern, the guard expressions are evaluated in left-to-right order. If a guard expression evaluates to #f, then it is as if the datum had failed to match against the pattern: the remaining guard expressions are ignored, and the next clause is tried. The expression (*fk*) is evaluated if the pattern it is associated with fails to match, or if the pattern matches but the guard fails.

ppat does the actual pattern matching over constants and pairs. The wild-card pattern _ matches against *any* value; the second pattern matches against the empty list; the third pattern matches against a quoted value; and the fourth pattern matches against *any* value, and binds that value to a lexical variable with the specified identifier name. The fifth pattern matches against a pair, tears it apart, and recursively matches the *car* of the value against the *car* of the pattern. If that succeeds, the *cdr* of the value is recursively matched against the *cdr* of the pattern. (We use **let** to avoid building long *car*/*cdr* chains.) The last pattern matches against constants, including symbols.

Here is the definition of h after expansion.

```
(define h
  (λ (x y)
    (let ((v `(,x . ,y)))
      (let ((fk (λ ()
                  (let ((fk (λ () (* x x))))
                    (if (pair? v)
                      (let ((vx (car v)) (vy (cdr v)))
                        (let ((c vy))
                          (if (number? c) (* c c) (fk))))
                      (fk))))))
        (if (pair? v)
          (let ((vx (car v)) (vy (cdr v)))
            (let ((a vx))
              (let ((b vy))
                (if (and (number? a) (number? b))
                  (+ a b)
                  (fk)))))
          (fk))))))
```

There are foun kinds of improvements that should be resolved by the compiler. First, *vx* is not used in the top definition of *fk*, so it should not get a binding. Second, the binding to a and b should be parallel **let** bindings. Third, where c is bound, could have been where *vy* is bound, and where a and b are bound, could have been where *vx* and *vy* are bound, respectively. Fourth, thunk creation is unnecessary where no guard is present.

The **mv-let** macro can be defined using **pmatch**.

```
(define-syntax mv-let
  (syntax-rules ()
    ((_ ((x ...) e) b0 b ...) (pmatch e ((,x ...) b0 b ...)))))
```

```
(mv-let ((x y z) (list 1 2 3))
  (+ x y z))
⇒ 6
```